Reusing Learned Information in SAT-based ATPG*

Görschwin Fey Tim Warode Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{fey,drechsle}@informatik.uni-bremen.de

Abstract

The robustness of engines for ATPG has to be improved to cope with the growing size of circuits. Recently, SAT-based ATPG approaches have been shown to be very robust even on large industrial circuits. Here, we propose techniques to further improve the efficiency by embedding learning techniques in a SAT-based ATPG engine. We provide a heuristic to apply incremental SAT when enumerating faults and a technique to apply circuit-based learning where incremental SAT is not applicable. The correctness of circuit-based learning is proven. Experimental results on large benchmarks show the efficiency.

1 Introduction

The size of circuits is steadily increasing. Especially for *Automatic Test Pattern Generation* (ATPG) this is a problem due to the NP-completeness of the task. The classical ATPG algorithms, like FAN [5] or PODEM [6], reach their limits. But for the post-production test of circuits, ATPG is essential.

In the recent past ATPG engines based on *Boolean satisfiability* (SAT) [9, 16] have shown their robustness even on large industrial circuits [14]. SAT-based engines are especially useful to classify faults that are hard for other engines, e.g. redundancies [13]. These tools mainly benefit from recent advances in the SAT domain. When applied to a SAT instance the SAT solver learns information each time a non-solution subspace is found [10]. Techniques to reuse this information for similar SAT instances have been proposed [18] and applied e.g. for bounded model checking [15, 7]. The challenge for reuse lies in the creation of the SAT instance and storing the learned information in a database. Domain specific knowledge is needed to allow for efficient reuse. Results for SAT-based ATPG have been reported for path delay faults [1], but in this case dynamic learning is based on the time consuming calculation of unsatisfiable cores. In classical ATPG algorithms learning was applied by considering non-trivial implications statically [12] or by detecting structural dependencies dynamically in a time consuming preprocessing step [8].

Here, we propose two strategies to reuse dynamically learned information for SAT-based ATPG of stuck-at faults. The first approach makes use of *incremental SAT* [18]. In this paradigm the SAT solver is never released, but the SAT instance is modified on the fly. So learned information is kept, if applicable. A heuristic to enumerate stuck-at faults such that subsequent SAT instances are very similar is proposed. The second approach applies a more general circuit-based learning scheme. This is necessary when SAT-based ATPG is applied in a multi-engine environment as it is usually done in industrial practice. The correctness of this learning approach is proven. Both techniques are applied to publicly available benchmark circuits and large industrial circuits. The experimental results show that the performance and robustness of SAT-based ATPG are significantly improved.

The paper is structured as follows: Preliminaries are provided in Section 2. In Section 3 the heuristic to apply incremental SAT is presented. Then, the approach for circuit-based learning is introduced and proven to be correct. Experimental results for a range of benchmark circuits are given in Section 5. Conclusions are presented in the last section.

2 Preliminaries

2.1 SAT

A *Conjunctive Normal Form* (CNF) is a set of clauses. A clause is a set of literals. A literal is either positive or negative. A positive literal is a variable, a negative literal is the negation of a variable.

A CNF is satisfied under an assignment of the variables if all clauses are satisfied. A clause is satisfied, if at least one literal is satisfied. A positive (negative) literal is satisfied, if the variable is assigned to 1 (0).

*This work was supported in part by Philips Semiconductors GmbH, Hamburg, Germany within the BMBF project MAYA and in part by DFG grant DR 287/15-1.

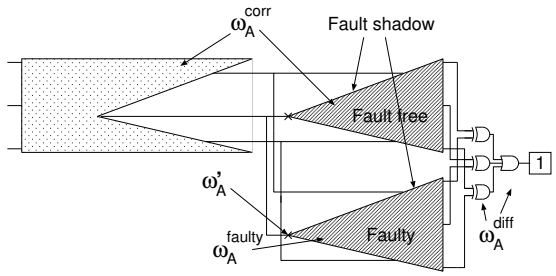

Figure 1. Structure of ϕ_A

A SAT solver searches for a satisfying assignment for a given CNF formula. In our paper conflict based learning as applied in modern SAT solvers is most important. Essentially, a backtrack search based on the DPLL procedure is performed [2]. Each time a conflict (a partial non-satisfying assignment) is found, this conflict is analyzed and a *learned clause* is added to the CNF to store the non-solution space.

2.2 Incremental SAT

Incremental SAT has been proposed to reuse learned information when a series of structurally similar SAT instances has to be solved [18]. Given two CNF formulas ϕ_A, ϕ_B and ϕ_A is solved first. Then, all clauses learned from $\phi_A \cap \phi_B$ can directly be applied when solving ϕ_B. Reusing the learned information speeds up the solving process for ϕ_B. Internally a SAT solver stores an implication graph. Therefore for any conflict clause c the set of clauses ϕ_c that implied c can be determined, i.e. $\phi_c \rightarrow c$. Then, c may be reused for ϕ_B if $\phi_c \subseteq (\phi_A \cap \phi_B)$. Completely storing ϕ_c for each conflict clause c is too inefficient. Therefore modern SAT solvers, like e.g. ZChaff [11], allow to define groups of clauses. Each conflict clause c has a tag that determines the groups that imply c. Only complete groups may be removed from the SAT instance. Then all clauses learned from this group are automatically removed as well. Adding clauses or groups of clauses is always allowed.

2.3 Circuits and Fault Modeling

A circuit $\mathcal{C}$ is composed of gates. The connections between the gates are described by a graph structure. A Boolean function is associated to each gate. The transitive fanin of a gate G is denoted by $\mathcal{F}(G)$. The transitive fanin of a set of gates $\mathbb{G}$ is denoted by $\mathcal{F}(\mathbb{G})$. The circuit can be transformed into a CNF ϕ by adding a set of clauses ω_G for each gate G: $\phi = \bigcup_{G \in \mathcal{C}} \omega_G$.

A *Stuck-At Fault* (SAF) occurs if a single line in a circuit is stuck at a constant value and does not depend on the values of the primary inputs any more. A test pattern for a SAF leads to different output values depending on the presence of the fault. If no test pattern exists, the SAF is redundant.

2.4 SAT-based ATPG

SAT-based ATPG was first proposed in [9]. Given a circuit and a SAF A, a SAT instance ϕ_A is created that is only satisfiable if a test pattern for A exists. If ϕ_A is unsatisfiable the fault is redundant. Figure 1 shows the structure of ϕ_A. Essentially, ϕ_A contains a model of the circuit without the fault and a model with the fault. Then the SAT solver searches for an input assignment that forces at least one output to different values in the two models. Parts of the models are shared. Constraints to model different parts of the SAT instance are denoted as follows:

- ω_A^{corr} – the gates in the correct model of the fault shadow or the shared part of the circuit
- ω_A^{faulty} – the faulty part of the circuit
- ω_A^{diff} – forces a difference at least at one output
- ω'_A – the faulty copy of the gate at A

Thus,

$$\phi_A = \omega_A^{corr} \cup \omega'_A \cup \omega_A^{faulty} \cup \omega_A^{diff}.$$

In the following for a gate G the constraints contained in ω_A^{corr} are always denoted by ω_G.

This is only a basic explanation of SAT-based ATPG. In practice additional constraints are added to the SAT instance to reflect the structure of the circuit [16, 13]. This makes SAT solving more efficient. For solving the SAT instance modern SAT solvers [10, 11, 4] are used and dedicated decision strategies are applied [13]. A more detailed presentation of SAT-based ATPG can be found in [3].

3 Incremental SAT for SAT-based ATPG

In the context of ATPG the enumeration of all faults determines the series of SAT instances considered. The order can be freely chosen. The objective is to enumerate faults such that subsequent SAT instances

1. have large identical parts and
2. the identities can be determined efficiently.

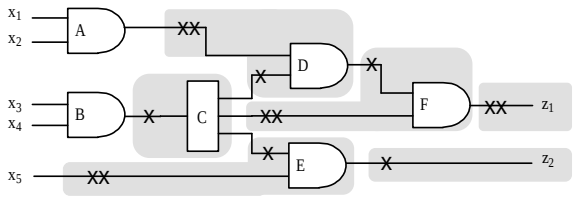

Figure 2. Example for gate-input-partitioning

We developed a heuristic to partition the set of faults. All faults in a single partition are handled incrementally. The clauses in the SAT instance are grouped depending on the heuristic. While enumerating faults in a single partition, some groups of clauses are kept while others are replaced. When continuing with the next fault partition, the whole SAT instance is rebuilt and all learned information is dropped.

As an extreme, each fault can be put into a separate partition. This corresponds to independent calls of the SAT solver for each fault. No information is reused. This partitioning is called *classic* in the following. On the other hand all faults can be stored in a single partition. Then, the fault free part of the circuit always contains a model of the whole circuit. Clauses resulting from this part are never dropped. Learned information is accumulated during ATPG, but may cause a significant overhead in the size of the SAT instance. This partitioning is called *totalInc*.

More promising is a compromise between the two extremes. In the following we show the *gate-input-partitioning* heuristic. A partition contains all faults at the inputs of a gate. An example for this partitioning is shown in Figure 2. Six partitions are created shown by gray boxes. Each 'x' denotes a SAF. Note that no fault collapsing is considered in the figure, whereas we applied fault collapsing as a preprocessing step in our experiments.

Given a gate G, large parts of the SAT instances that correspond to faults in a single partition are identical:

- Fault shadow: Due to the use of fanout gates, all fault locations have the same paths to the primary outputs. Therefore the fault shadow is identical for all faults in the partition. This is valid for the faulty part and the fault free part of the circuit.
- Fault free part of the circuit: The fault free part contained in the SAT instance is determined by traversing the circuit from outputs in the fault shadow towards inputs. Because the fault shadow is identical, also the fault free part of the circuit is identical.

```
,
for each faultpartition do {
  extractClauses( globalGroup );
  for each fault in faultpartition do {
    extractFaultyGate( faultGroup );
    extractFaultSite( faultGroup );
    solve();
    deleteClauses( faultGroup );
    resetSatSolver();
  }
 deleteAllClauses();
}
```

Figure 3. Algorithm based on gate-input-partitioning

All clauses corresponding to these parts, i.e. $\omega_A^{corr} \cup \omega_A^{faulty}$, are summarized in the group *globalGroup* of clauses.

The only difference between two SAT instances is the model of the gate that is considered faulty. Different clauses are needed to model the stuck-at value at different inputs of the gate. Also the two SAFs at a single input differ in their value. Therefore all clauses to model the gate and the fault value (ω'_A) are collected in the group *faultGroup*.

The overall ATPG algorithm for gate-input-partitioning is shown in Figure 3. All partitions are enumerated. The function *extractClauses(globalGroup)* creates the clauses in $\omega_A^{corr} \cup \omega_A^{faulty}$. These clauses are stored in *globalGroup* and are not changed while enumerating other faults in the current partition. Then, all faults within the partition are handled individually. The clauses to encode ω'_A, i.e. to model the faulty gate (*extractFaultyGate*) and the fault value (*extractFaultSite*), are created and stored in *faultGroup*. By solving the SAT instance, the function *solve* classifies the fault. Afterwards, all clauses in *faultGroup* and all clauses derived from this group are removed by calling the procedure *deleteClauses*. Finally, to restart the search the SAT solver has to be reset before proceeding to the next fault. Only if a new partition is considered all clauses are removed.

Besides gate-input-partitioning other heuristics were implemented and evaluated, e.g. by grouping faults along paths, at outputs or a combination of these heuristics [17]. But gate-input-partitioning turned out to be the most efficient partitioning scheme in our experiments regarding run time and memory consumption.

4 Enhanced Circuit-based Learning

In practice there is usually a large number of easy to detect faults that can be classified very efficiently by random simulation. More sophisticated algorithms, like FAN or PODEM, efficiently classify harder faults. SAT-based approaches are well suited for hard to detect faults and also perform very well in case of redundancies. For this reason, a state of the art ATPG system applies multiple engines where individual faults are passed to particular engines. The choice of the engine is usually done by heuristics. As a result, statically partitioning all faults during preprocessing is not feasible. Instead, learning has to be circuit-based and must be independent from the SAT instance and the ATPG engine. We provide an efficient circuit-based learning strategy and prove the correctness of the approach.

First, learned clauses are stored in a database, then stored clauses are considered for reuse. In the database a learned clause c is stored as a set of literals $\{l_0, \ldots, l_n\}$. A variable in the SAT instance corresponds to the output of a gate. Therefore each literal is a pair (G_i, P_i) where G_i denotes the gate and P_i the polarity, $P_i = 0$ denotes the negative literal, $P_i = 1$ denotes the positive literal. After solving a SAT instance the learned clauses are analyzed and stored in the database if the following rule applies.

Rule: Clause c is derived from the fault free part of the circuit, i.e. $\omega_A^{corr} \rightarrow c$.

There are two reasons to apply this rule. First, the precondition can be evaluated efficiently across all SAT instances for different faults. More detailed, when all clauses in ω_A^{corr} are summarized in a single group the decision whether the clause can be derived solely from the fault free part of the circuit is easy. Second, clauses derived only from the fault free part can be reused more easily than clauses derived from the faulty part of the circuit where the injected fault changes the functionality. Note that for efficiency in practice only those clauses are stored that have three literals or less.

The next step is the reuse of stored clauses. Inserting a stored clause c into a SAT instance ϕ_A is only allowed if $\phi_A \rightarrow c$. This check has to be carried out efficiently because it is done for each fault and each stored clause. We provide such an efficient check and prove the correctness: in our context it is sufficient to check whether ϕ_A contains clauses for all gates that are considered in c. Two lemmas are used to prove the main result.

Lemma 1. *Let ϕ_A be a SAT instance for SAF A and for gate G let $\omega_G \subseteq \phi_A$. Then, for any gate H in the transitive fanin $\mathcal{F}(G)$ of G also $\omega_H \subseteq \phi_A$.*

Proof. Due to construction $\omega_G \subseteq \omega_A^{corr} \subseteq \phi_A$. Constraints for gate G are only inserted if G is reached while traversing the circuit towards the primary inputs. Then, constraints for all gates in $\mathcal{F}(G)$ are also inserted into ϕ_A. □

Lemma 2. *Let $c = \{l_1, \ldots, l_n\}$ be a stored clause, ϕ_A be a SAT instance for SAF A, and $\phi_A \rightarrow c$. Let $\mathbb{G} = \{G : (G, P) \in c, \text{where } P \in \{0, 1\}\}$. Then c can be implied by considering only $\phi_c = \cup_{H \in \mathcal{F}(\mathbb{G})} \omega_H$, i.e. all clauses coming from gates in the fanin of $\mathbb{G}$.*

Proof. According to the rule for storing clauses it is sufficient to consider ω_A^{corr}. Due to construction $\phi_c \subseteq \omega_A^{corr}$.

Given the values of all but one gate the value of the last gate can be implied. Therefore the clause c can be rewritten as $\{\bar{l}_1, \ldots, \bar{l}_{n-1}\} \rightarrow l_n$ (any other literal than l_n may be chosen, but choosing n simplifies the notation). The value of a gate G only depends on its predecessors in the circuit, i.e. on $\mathcal{F}(G)$.

Let α be a CNF that is only satisfied by an assignment to the primary inputs that forces all gates $G_i, i \leq n$ to the values $\overline{P}_i$. If no such assignment exists, $\phi_c \rightarrow c$ holds (because $\bar{l}_1 \ldots \bar{l}_{n-1}$ is never satisfied).

Otherwise $\phi_c \cup \alpha$ can only be satisfied under a variable assignment if G_n assumes the value P_n (because $\bar{l}_1 \ldots \bar{l}_{n-1} \rightarrow l_n$ holds on ω_A^{corr}). Thus, $(\phi_c \cup \alpha) \rightarrow l_n$ holds. By construction the constraint α is equivalent to $\bar{l}_1 \ldots \bar{l}_{n-1}$ with respect to ϕ_c. Thus $(\phi_c \cup \{\bar{l}_1, \ldots, \bar{l}_{n-1}\}) \rightarrow l_n$. Therefore, if $\bar{l}_1 \ldots \bar{l}_{n-1}$ is satisfied, ϕ_c can only be satisfied if l_n is satisfied. This leads to $\phi_c \rightarrow c$. □

Theorem 1. *Let $c = \{l_1, \ldots, l_n\}$ be a stored clause and ϕ_A be a SAT instance for SAF A. Furthermore for each $i \in \{1, \ldots n\}$ and $l_i = (G_i, P_i)$ let $\omega_{G_i} \subseteq \phi_A$. Then, $\phi_A \rightarrow c$.*

Proof. Clause c was learned previously on a SAT instance ϕ_B for SAF B. According to Lemma 2, clause c can be implied by ϕ_c (as defined in the lemma). Furthermore $\phi_c \subseteq \phi_A$ according to Lemma 1. Thus, $\phi_A \rightarrow c$. □

Based on this foundation, we propose two learning approaches. First we applied learning only in a preprocessing step. For each output the circuit is converted into a CNF and the SAT solver is started on this CNF. The learned clauses of this run are considered for creating a *static* database. The second approach applies *dynamic* learning. After running the SAT solver on the SAT instance for a particular fault the database is updated with the learned clauses.

Table 1. Run time for incremental SAT

circ	classic		gate-input			totalInc	
	eqn	sat	eqn	sat	imp.	eqn	sat
c432	3.0	1.4	1.3	1.3	1.69	6.3	6.1
c499	10.0	54.6	4.7	35.0	1.63	30.5	61.0
c1355	17.4	83.7	6.6	43.5	2.02	45.7	86.1
c1908	13.2	15.9	5.8	12.5	1.59	45.6	51.7
c3540	49.4	37.7	20.2	31.4	1.69	167.5	157.0
c7552	102.2	130.6	46.7	93.3	1.66	449.5	536.3
s1494	2.1	1.7	1.0	1.7	1.41	8.4	10.1
s5378	19.5	7.6	8.7	5.5	1.91	111.9	132.7
s15850	145.6	70.9	66.8	58.6	1.73	1693.5	1.318.7
s38417	220.0	88.1	95.8	70.8	1.85	mem. out	
b10_C	0.5	0.2	0.2	0.1	2.33	1.2	1.0
b11_C	6.4	2.2	2.8	1.8	1.87	19.6	20.8
b12_C	6.8	3.3	2.8	2.7	1.84	47.8	51.6
b14_C	856.9	2485.1	391.7	1921.2	1.44	mem. out	
b15_C	1310.9	4511.9	555.0	3432.5	1.46	mem. out	
				avg	1.74		

Table 2. Run time of learning on top of gate-input-partitioning

circ	gate-inp.	static		dynamic	
	time	time	imp.	time	imp.
c432	2.6	2.7	0.96	2.6	1.00
c499	39.7	30.7	1.29	21.0	1.89
c1355	50.1	40.0	1.25	32.5	1.54
c1908	18.3	16.9	1.08	14.4	1.27
c3540	51.6	54.1	0.95	47.9	1.07
c7552	140.1	145.6	0.96	106.5	1.31
s1494	2.7	2.7	1.00	2.8	0.96
s5378	14.2	15.5	0.91	14.3	0.99
s15850	124.4	139.3	0.89	121.3	1.02
s38417	166.6	191.3	0.87	226.0	0.73
b10_C	0.3	0.4	0.75	0.3	1.00
b11_C	4.6	4.8	0.95	5.1	0.90
b12_C	5.5	5.6	0.98	5.6	0.98
b14_C	2312.9	1982.6	1.16	1426.8	1.62
b15_C	3987.5	3665.3	1.08	2673.6	1.49
		avg	1.00	avg	1.18

5 Experimental Results

In the experiments we consider benchmark circuits from the ISCAS85, ISCAS89 and ITC99 benchmark sets as well as industrial benchmarks from Philips Semiconductors GmbH, Hamburg, Germany. All experiments were carried out on an AMD Athlon XP 64 3500+ system (Linux, 2200 MHz, 1 GB). Due to page limitation only a subset of the results can be reported in the following. The proposed learning techniques are implemented on top of the SAT-based ATPG tool PASSAT that often outperforms classical ATPG algorithms [13]. Results of the application of PASSAT to industrial circuits were already reported [14]. PASSAT applies a four-valued logic to handle multi-valued benchmarks such as tri-state values and unknown values coming from the environment of the circuit. We use the SAT solver ZChaff [11] in the 2004 version that provides an interface for incremental SAT. For a circuit all SAFs are classified using the SAT-based engine. No other engines and no fault simulation are applied (which can further speed up ATPG in practice). Fault collapsing is used to reduce the number of faults in advance. For each remaining fault a time out of 20 seconds was applied, otherwise the classification was aborted. Additionally, the proposed learning techniques were embedded.

Results for the application of incremental SAT are shown in Table 1. Data is presented for the partitioning *classic*, *gate-input* and *totalInc* as explained in Section 3. For each algorithm the total run times for generating the SAT instances (*eqn*) and solving (*sat*) are reported in seconds. The speed-up of gate-input-partitioning vs. classic is also reported (*imp*). Even *classic* classified all faults within the time out, i.e. no aborts occurred. Compared to the classical approach gate-input-partitioning provides remarkable speed-ups. The generation of the SAT instances is done much faster because large parts are simply reused. Also the time for solving the problems is significantly reduced due to the learned clauses. On average a speed-up of 1.74 was obtained on the benchmarks. The memory needs for gate-input-partitioning were the same as for the algorithm *classic*. In contrast, *totalInc* causes a drastic increase in memory usage due to a large number of learned clauses that were accumulated while enumerating all faults. As a result even the run time increased and in some cases the memory limit of 1250MB (including swapping space) was exceeded.

Next, we applied the two circuit-based learning approaches to the classical algorithm without incremental SAT and to the algorithm based on gate-input-partitioning. In both cases learning improved the overall run time. Experimental results for the combination with gate-input-partitioning are reported in Table 2. Here, the improvements are reported in comparison to gate-input-partitioning without learning. When gate-input-partitioning is used, the preprocessing does not improve the overall performance. The learned clauses stem from "simple" conflicts and do not improve the performance for hard SAT instances. In contrast the dynamic approach that analyzes and stores learned clauses after each run of the SAT solver improves the performance on average by another 18% over gate-input-partitioning, i.e. by 217% over *classic*. This shows that especially reusing learned clauses from hard faults helps to improve the overall perfor-

Table 3. Results for industrial circuits

		classic		gate-inp+dynamic	
circ	#faults	ab.	time	ab.	time
p77k	126,338	0	4,487	0	3,832
p80k	181,160	12	24,703	0	12,116
p88k	133,891	2	13,572	0	5,755
p99k	140,633	63	26,343	19	15,397
p177k	260,812	6,974	372,546	236	95,452
p462k	616,735	6,232	309,828	19	62,921
p565k	1,317,213	4,306	495,167	540	284,235
p1330k	1,441,878	132	166,791	14	221,060

mance. Note, that *all* possible faults in the circuits where classified by the SAT-based approach. But the overhead of generating a SAT instance only pays off for faults that are hard to classify. In our case this overhead occurs even for the large number of "easy-to-detect" faults that could be classified much more efficiently by random simulation. Therefore the overall run time could not be improved in some cases.

Finally, results for industrial benchmark circuits are reported in Table 3. The name of a circuit also gives the number of gates contained in the circuit, e.g. p565k has about 565.000 gates. The number of faults after collapsing is reported in the second column. The classical algorithm without learning is compared to the algorithm that combines gate-input-partitioning with dynamic learning. The number of faults that were aborted are reported in column *ab*. Column *time* give the total run time. The results show that the learning techniques significantly improve the robustness of SAT-based ATPG. A large number of faults was aborted by the classical algorithm. In contrast only a few aborted faults remain when learning is applied. Moreover, even the run time decreases in most cases. In one case the improvement even reaches a factor of 4.9. The runtime was only increased for p1330k, but at the same time the number of aborted faults was reduced significantly. This shows that storing learned information is essential to classify hard faults.

Overall the performance of SAT-based ATPG can be significantly improved. Especially the combination of gate-input-partitioning and dynamic circuit-based learning boosts robustness. The run time is reduced on average and the number of aborted faults is reduced for all benchmarks considered.

6 Conclusions

We presented a SAT-based ATPG engine with embedded learning strategies. Both paradigms, i.e. incremental SAT and circuit-based learning, were exploited. For the more difficult case of circuit-based learning the correctness of the technique was proven. Experimental results show an improved robustness on large industrial benchmarks.

The next step is the tight integration with classical ATPG engines. In this context the SAT-based tool can be used to efficiently handle faults that are hard to classify using other techniques. By reusing learned information for the other engines, e.g. FAN, the overall performance can be further improved.

Additionally, the extension of SAT-based ATPG to other fault models, such as the path delay fault model or the bridging fault model, is considered.

References

[1] K. Chandrasekar and M. S. Hsiao. Integration of learning techniques into incremental satisfiability for efficient path-delay fault test generation. In *Design, Automation and Test in Europe*, pages 1002–1007, 2005.

[2] M. Davis, G. Logeman, and D. Loveland. A machine program for theorem proving. *Comm. of the ACM*, 5:394–397, 1962.

[3] R. Drechsler and G. Fey. Automatic test pattern generation. In *Formal Methods for Hardware Verification*, LNCS, pages 30–55, 2006.

[4] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[5] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. on Comp.*, 32:1137–1144, 1983.

[6] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic. *IEEE Trans. on Comp.*, 30:215–222, 1981.

[7] D. Große and R. Drechsler. Acceleration of SAT-based iterative property checking. In *CHARME*, volume 3725 of *LNCS*, pages 349–353. Springer, 2005.

[8] W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *Int'l Conf. on CAD*, pages 538–543, 1993.

[9] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Trans. on CAD*, 11:4–15, 1992.

[10] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

[11] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.

[12] M. Schulz, E. Trischler, and T. Sarfert. SOCRATES: A highly efficient automatic test pattern generation system. In *Int'l Test Conf.*, pages 1016–1026, 1987.

[13] J. Shi, G. Fey, R. Drechsler, A. Glowatz, F. Hapke, and J. Schlöffel. PASSAT: Effcient SAT-based test pattern generation for industrial circuits. In *IEEE Annual Symposium on VLSI*, pages 212–217, 2005.

[14] J. Shi, G. Fey, R. Drechsler, A. Glowatz, J. Schlöffel, and F. Hapke. Experimental studies on SAT-based test pattern generation for industrial circuits. In *Int'l Conf. on ASIC*, pages 967–970, 2005.

[15] O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME*, volume 2144 of *LNCS*, pages 58–70, 2001.

[16] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Trans. on CAD*, 15:1167–1176, 1996.

[17] T. Warode. Strukturelles Lernen in der erfüllbarkeitsbasierten Testmustergenerierung (Structural learning for test pattern generation based on satisfiability). Master's thesis, University of Bremen, 2006.

[18] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Design Automation Conf.*, pages 542–545, 2001.